Secure Processor Architecture for High-Speed Verification of Memory Integrity

ATSUYA OKAZAKI,† MASAKI NAKANISHI†† and SHIGERU YAMASHITA††

This paper proposes a processor-integrated approach to verify the data integrity of external memory. Present processors always trust the data in off-chip memory without question. However, assuring the data integrity of external memory will be one of the important basic technologies in secure computing environments in the future. A processor with the proposed mechanism can guarantee the integrity against memory corruption even when running malicious software (e.g., computer viruses, corrupted untrusted operating systems) or when subject to physical attacks.

Many software verification methods have been proposed. However, those methods have limitations in speed and security in comparison with hardware-integrated approaches. A processor architecture that uses an "Incremental Multiset Hash Function" has also been proposed by G. E. Shu et al. However it degrades the processor speed due to its heavy use of hash operations.

Our architecture achieves high-speed verification of memory integrity using a hardware accelerator that monitors the behavior of memory accesses. By using a complicated but computationally lightweight scheme to count memory accesses, the architectures can use Rijndael symmetric key cryptography, which is a superset of AES and has fast implementations, instead of a computationally expensive hash function. That contributes to high-speed operation of integrity verification. The speed was evaluated by using SimpleScalar. The instructions per cycle of our architecture were consistently better than in previous approaches, and improved by 60% for high cache miss rates.

1. Introduction

As Internet e-commerce, digital content distribution and so on have wide spread, security systems for personal information protection or digital right management have become more important. On the other hand, there have been increasing amounts of malicious software (e.g., computer viruses) and increasing numbers of physical attacks on hardware chips or buses, where the security of the systems can be threatened. Thus, most of today's security countermeasures relying only on software are becoming more and more vulnerable to untrusted operating systems attacked by malicious software or against physically attacked hardware.

Therefore, some security infrastructures using not only software but also hardware that can safeguard program execution have recently been proposed to create more secure computing environments. They are trying to provide a secure computing environment from a low level position under software by using tamper-resistant of hardware and high-speed implementations of cryptographic functions. For example, the Next Generation Secure Computing Base[1] and LaGrande[2] with the Trusted Platform Module[3] were proposed by Microsoft, Intel, and the Trusted Computing Group for authenticated program execution and digital rights management. The eXecute Only Memory architecture [4),5)] was proposed against protect software privacy, where encrypted programs can only be executed on a processor that has a certain secret key.

Among the fundamental technologies used secure infrastructures which are integrated with both software and hardware, we focus on the data integrity of off-chip memory to assure that program and data stored in external memory are not corrupted. In this paper, we propose a secure processor architecture to assure the integrity of memory even under untrusted operating systems or physical attacks direct toward the hardware. In cases that require absolute security such as e-commerce, digital content distribution, electronic voting, and so on, a processor with the proposed mechanism can provide a highly secure computing environment.

A secure processor, AEGIS, proposed by Suh et al.[6)~8)] also has a mechanism to assure the integrity of memory. Data and address transactions on an off-chip memory bus and their sequences are monitored by the processor, and are compressed into a hash value by using an Incremental Multiset Hash Function (IMHF)[9]. Their architecture executes a heavy hash function SHA-1[10] for every memory access, and thus the speed of the processor is decreased.

Our secure processor architecture uses a computationally lightweight special counting scheme for memory access, and achieves high-speed in verifying the integrity of memory. By using this novel counting scheme, the fast block cipher Rijndael[11] can be used instead of an IMHF in the previous work. This contributes to the high-speed integrity verification.

This paper is organized as follows. Section 2 formulates the problem of off-chip memory integrity verification. Section 3 introduces the proposed method. Section 4 describes the evaluated results of the method with a comparison to the previous approaches using a hardware simulation. Our conclusions are in Section 5.

2. Problem Formulation and Goal

This section formulates the problem of off-chip memory integrity, and defines the goals of the proposed method.

We assume that a main memory (e.g., DRAM) is separated from the processor. Such processors generally have cache memory inside the chips (See Figure 1). The in-

† IBM Research, Tokyo Research Laboratory
†† Nara Institute of Science and Technology

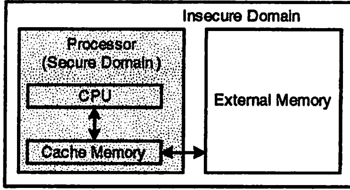

Fig. 1 The architecture model of the supposed system.

side of the processor is regarded as a secure domain that has tamper resistance against malicious software or physical attacks. An adversary cannot access secret information in the processor, but the external components and the interfaces of a processor are not protected against an adversary. Data and address bus and control signals can be observed. It is also assumed that an adversary can replace existing data on the off-chip memory with other data at any time. For example, if the adversary uses physical equipment such as a logic analyzer or a memory emulator that is connected to the memory bus, the adversary can perform such attacks.

We define memory integrity as follows: "Memory behavior is valid if the data read from a particular address is the same as the data written most recently to that address." When the proposed processor is requested to guarantee its integrity, the processor switches to "secure-mode." If the data in the memory has been altered by an adversary, the processor can detect the attack by the end of the process in the secure-mode.

The goal of the proposed architecture is to insure the integrity of memory. In addition, we aim to reduce the verification overhead and to provide a high-speed memory integrity verification method.

3. The Proposed Method

The proposed architecture contains several nontrivial ideas, and thus it is relatively difficult to understand with a description of the architecture alone. Therefore, we start with an intuitive description of the behavior of the method in Section 3.1 before we go into the details in Section 3.2. Note that the discussion in Section 3.1 lacks technical completeness, but it is still helpful as an overview of the proposed method.

3.1 Overview

This section introduces a skeletal overview of the proposed method. If the data read from external memory is the same as the data previously written to the same address, the data is valid. A processor, therefore, can guarantee the integrity by verifying the data transactions. Most modern processors have cache memories, and therefore, it is sufficient to verify whether the data previously written-back to the external memory is the same as the data refilled from that memory whenever a cache replacement occurs (See Figure 2). The proposed processor has a verification module between the cache memory and the external memory bus to verify those transactions.

In our approach, while in the secure-mode, the read and

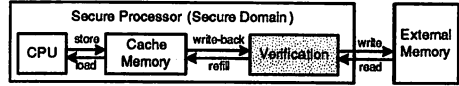

Fig. 2 The verification module in the secure processor.

Fig. 3 Compressing transaction history for each read/write channel.

write transactions are independently XORed with read and write history registers in the processor in order to generate transaction histories, and stored into these registers again, respectively. However, if the raw data of the transactions is XORed, an adversary can easily determine the values of the history registers. Therefore, the read and write transactions are independently encrypted with the block cipher Rijndael and the encrypted data is sequentially XORed into the read and write history registers (See Figure 3). At the end of process in the secure-mode, if the read and write history registers hold same values, it is assured that the memory has not been corrupted. If the registers hold different values, it means corruption has occurred.

For any cache operation, a pair of read and write accesses to the external memory are executed so that the history of the read transaction will be the same as that of the write transaction for that memory address. That is, even when the cache operation is a write-back operation, the processor first reads the data from the external memory and updates the read history register with the read data. Next, the processor writes the new data to the memory and updates the write history register with the written data. Similarly, when the cache operations is refill, the processor first reads data from the memory and updates the read history register. Next, the processor writes the read data to the memory and updates the write history register. When an illegal write access to the memory from outside of the processor is occurred, these history registers are not updated by using the valid XOR operations, and thus this causes a mismatch between the values in the read and write history registers.

However, a reordering of the input arguments of XOR operations, or even-numbered XOR operations of the same data do not effect the final value of XOR operations (e.g., $a \oplus b = b \oplus a$ or $a = a \oplus a \oplus a$. Detailed examples of these are shown in the following section). Therefore, an adversary may falsify the sequence and the number of memory accesses by using these characteristics, and may obliterate the history of attacks. In order to detect such attacks, AEGIS uses the IMHF, including a heavy hash function. However, our new architecture can also detect such attacks in high-speed. The proposed architecture includes memory access counters, that write the number of memory accesses to the external memory. But an adversary may also rewrite the counter value to erase the sequence of illegal memory accesses. In order to prevent

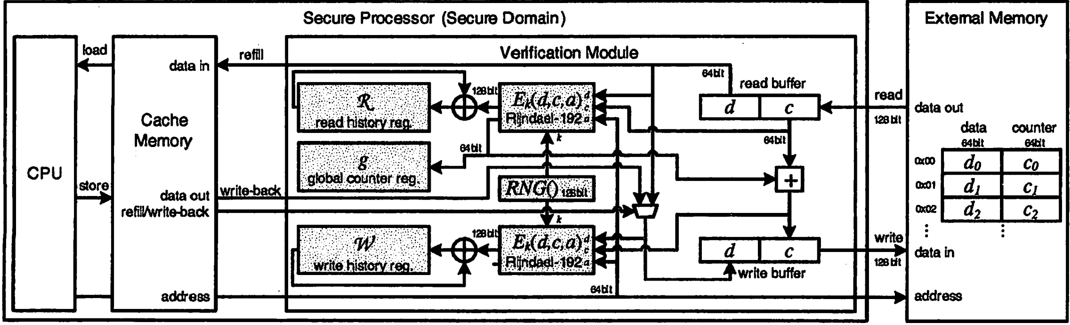

Fig. 4 The hardware architecture for memory integrity verification.

this, the counter value is encrypted with Rijndael. This Rijndael circuit can be shared with the circuit that generates the transaction history to reduce the circuitry requirements.

3.2 The Architecture and Behaviors of the Verification Module

This section explains the details of the architecture and its behavior. In general, a processor stores only data into external memory. However, the proposed secure processor stores both data and the value of the memory access counter during the secure-mode. The way to update the memory access counter is explained belows:

Figure 4 shows the architecture of our proposed processor-integrated verification module that consists of the components described below.

- **Random Number Generator** *RNG*(): When a new process starts in secure-mode, a 128-bit random number k is generated as a secret key for Rijndael.
- **Rijndael-192:** $E_k(d, c, a)$: A 192-bit (3×64-bit) data block (d, c, a) is encrypted by using a random number k generated by the random number generator. Then a 192-bit cipher text is output, where d, c, and a denote the data, the counter, and the address, respectively.
- **Read and Write history register** $\mathcal{R}, \mathcal{W}$: According to Equations (1) and (2), two 128-bit registers are updated at each memory access with the higher 128 bits of the Rijndael-192 output. The lower 64 bits are for the memory access counter. Subscripts r and w denote a read block and a write block, respectively. Subscripts H and L denote higher bits and lower bits, respectively. The time values t and $t-1$ denote the present and previous steps, respectively. The update operations can be described as follows:

$$\mathcal{R}_t := \mathcal{R}_{t-1} \oplus E_k(d_r, c_r, a_r)_H, \quad (1)$$

$$\mathcal{W}_t := \mathcal{W}_{t-1} \oplus E_k(d_w, c_w, a_w)_H,$$

$$\text{where } c_w = c_r + E_k(d_r, c_r, a_r)_L. \quad (2)$$

- **Global counter register** g: In order to detect deception utilizing the characteristics of the XOR operation as mentioned earlier, this register is incremented by the output of Rijndael as follows:

$$g_t := g_{t-1} + E_k(d_r, c_r, a_r)_L. \quad (3)$$

The verification module has four behaviors: **init**, **refill**, **write-back**, and **verify** (Detailed in Figure 5). The **init** initializes the whole memory space that should be assured for integrity at the beginning of the secure-mode. The **refill** and **write-back** generate a transaction history for each cache memory operation while in the secure-mode. These behaviors generate a pair of memory accesses (i.e., read and write accesses) for each operation. However, it should be noted that this overhead is considered to be reasonable for assuring the memory integrity. The **verify** reads the whole memory space and verifies the integrity at the end of the secure-mode. The verification module assures the memory integrity by using these behaviors. Examples of those behaviors are explained in the following section.

3.3 The special counting scheme

This section first describes three behavioral examples of the proposed architecture. The first example is a case verified as valid. The next two examples are cases for attacks using the characteristics of the XOR operation. Then the novel computationally lightweight counting scheme to block such attacks is explained.

Figure 6 shows three behavioral examples where the external memory has only one address for simplicity. (1) is a case verified as valid. (2) and (3) are the cases of attacks by using reordering or two XOR operations, respectively.

In Figure 6 (1), first the memory is initialized with 0 data, and the write history register is updated. Both read and write accesses are executed anytime a cache operation occurs, and the read and write history registers are updated. Finally the memory is read to update the read history register, and the final values of the read and write registers are compared. If an adversary alters data in the memory, these values will become inconsistent.

The above mechanism can detect most attacks, such as a simple attack that simply corrupts data in external memory, a spoofing attack that brings data stored in another address to overwrite a target address, and a replay attack that overwrites current data with the data that existed earlier in the same address. However the following two kinds of attacks cannot be detected. In Figure 6 (2), an adversary reorders the transactions to obliterate the history of the corruption. In Figure 6 (3), an adversary makes the processor read the corrupted b twice to obliterate the history of the corruption with two XOR operations. Therefore, the proposed architecture provides countermeasures

- **init**: (at the beginning of the secure-mode)
  write $\{d, c\}$ denotes writing a block that has d in the data field and c in the counter field.
  1. $k := RNG()$
  2. $\mathcal{R}_0 := 0, \mathcal{W}_0 := 0, g_0 := 0$
  3. $\forall a_n$ (for the whole specified secure address a_n)
     - write $\{0, E_k(0, 0, a_n)_L\}$ to a_n
     - $\mathcal{W}_t := \mathcal{W}_{t-1} \oplus E_k\big(0, E_k(0, 0, a_n)_L, a_n\big)_H$
     - $g_t := g_{t-1} + E_k(0, 0, a_n)_L$
- **refill**(a): (run-time operation in the secure-mode)
  1. read $\{d, c\}$ from a
  2. $\mathcal{R}_t := \mathcal{R}_{t-1} \oplus E_k(d, c, a)_H$
  3. write $\{d, c + E_k(d, c, a)_L\}$ to a
  4. $\mathcal{W}_t := \mathcal{W}_{t-1} \oplus E_k\big(d, c + E_k(d, c, a)_L, a\big)_H$
  5. $g_t := g_{t-1} + E_k(d, c, a)_L$
- **write-back**(d_{new}, a): (run-time operation in the secure-mode)
  1. read $\{d_{\text{old}}, c\}$ from a
  2. $\mathcal{R}_t := \mathcal{R}_{t-1} \oplus E_k(d_{\text{old}}, c, a)_H$
  3. write $\{d_{\text{new}}, c + E_k(d_{\text{old}}, c, a)_L\}$ to a
  4. $\mathcal{W}_t := \mathcal{W}_{t-1} \oplus E_k\big(d_{\text{new}}, c + E_k(d_{\text{old}}, c, a)_L, a\big)_H$
  5. $g_t := g_{t-1} + E_k(d_{\text{old}}, c, a)_L$
- **verify**: (at the end of the secure-mode)
  c_{a_n} denotes the final value in the counter field on memory address a_n. A denotes a set of specified secure addresses.
  1. $\forall a_n$
     - read $\{d_{a_n}, c_{a_n}\}$ from a_n
     - $\mathcal{R}_t := \mathcal{R}_{t-1} \oplus E_k(d_{a_n}, c_{a_n}, a_n)_H$
  2. $(g_{\text{final}} \stackrel{?}{=} \sum_{a \in A} c_{a_n})$ && $(\mathcal{W}_{\text{final}} \stackrel{?}{=} \mathcal{R}_{\text{final}})$

Fig. 5 The behaviors of the verification module.

write sequence →

g global counter register		0	1	2	3	4	5
c counter fields in external memory	address	value					
	0x00	0	0+1	1	1+1	2	2
	0x01	0	0	0	0	0+1	1
	0x02	0	0	0+1	1	1	1+1

(Final column sums to 5.) Write access is executed on the shaded block.

Fig. 7 An example of the global counter register and the counter fields.

by using a special counting scheme for memory accesses.

Each address in the external memory has a counter field c in addition to the data field d (See the external memory in Figure 4). The architecture counts memory accesses for each address by using the counter fields. The value of the counter field is included in the transaction history. Thus, reordering attacks can be detected. However, if an adversary could predict future values of the counter in some way, he could still reorder the transactions. Therefore, the counter value is encrypted to prevent such an attack.

The architecture also provides a processor-internal global counter register. Figure 7 shows a behavioral example of the global counter and each memory access counter field in the external memory, which has only three addresses (0x00-0x02) to simplify the explanation. Each counter field holds the number of write accesses to its address. The global counter counts all of the write accesses. At the end of the process in secure-mode, the sum of all of the counter fields must be equal to the global counter if no attack has occurred. Since the counter fields are included in the transaction history held within the processor, the counter field cannot be corrupted. In Figure 7, the values of the global counter register and the counter field increase by one at the same time to simplify the explanation. In a practical proposed architecture, the increments are the lower 64 bits of the Rijndael-192 output for a read channel.

In AEGIS, a sequence of transactions is also input into the IMHF using a heavy SHA-1 that has collision resistance, which can therefore prevent attacks using reordering and two XOR operations. Instead of such a heavy function, we adopted an architecture where the data encrypted by a high-speed block cipher is summed up by XOR operations, as represented in Equations (1) and (2). However, this simple scheme is not enough for the attacks using reordering and two XOR operations, thus, we also include a special counting scheme of memory access to prevent these attacks. This novel idea contributes to the high-speed integrity verification.

4. Evaluation

We used SimpleScalar v3.0d[12)] with the parameters shown in Table 1 for the performance evaluation. The verification module was located between the L2ID cache

(1) case verified as valid

Operation	$\mathcal{W}$ write history register	$\mathcal{R}$ read history register	Ext. Memory
init	$E_k(0)$		→ 0 (write)
write-back(a)	$\oplus E_k(a)$	$E_k(0)$	← 0 (read) → a (write)
refill	$\oplus E_k(a)$	$\oplus E_k(a)$	← a (read) → a (write)
verify		$\oplus E_k(a)$	← a (read)
Final value	$E_k(0)\oplus E_k(a)\oplus E_k(a)$	$E_k(0)\oplus E_k(a)\oplus E_k(a)$	

(2) case with reordering

Operation	$\mathcal{W}$ write history register	$\mathcal{R}$ read history register	Ext. Memory
init	$E_k(0)$		→ 0 (write)
write-back(a)	$\oplus E_k(a)$	$E_k(0)$	← 0 (read) → a (write)
		corruption	$a \to b$
refill	$\oplus E_k(b)$	$\oplus E_k(b)$	← b (read) → b (write)
		corruption	$b \to a$
verify		$\oplus E_k(a)$	← a (read)
Final value	$E_k(0)\oplus E_k(a)\oplus E_k(b)$	$E_k(0)\oplus E_k(b)\oplus E_k(a)$	

(3) case with two exclusive-ORs

Operation	$\mathcal{W}$ write history register	$\mathcal{R}$ read history register	Ext. Memory
init	$E_k(0)$		→ 0 (write)
write-back(a)	$\oplus E_k(a)$	$E_k(0)$	← 0 (read) → a (write)
		corruption	$a \to b$
write-back(a)	$\oplus E_k(a)$	$\oplus E_k(b)$	← b (read) → a (write)
		corruption	$a \to b$
write-back(a)	$\oplus E_k(a)$	$\oplus E_k(b)$	← b (read) → a (write)
verify		$\oplus E_k(a)$	← a (read)
Final value	$E_k(0)$~~$\oplus E_k(a)\oplus E_k(a)$~~$\oplus E_k(a)$	$E_k(0)$~~$\oplus E_k(b)\oplus E_k(b)$~~$\oplus E_k(a)$	

Fig. 6 The examples of valid and invalid accesses.

Table 1 The parameter settings for the simulation.

Processor model	Speculative out-of-order Alpha
L1I, L1D	64 KB 2-way 32 B, 64 KB 2-way 32 B
L1 latency	2 cycles
L2ID	Unified, 256 K - 1 MB - 4 MB, 64B
L2 latency	10 cycles
Memory latency (first, successive)	Normal (18, 2), Proposed (40, 2), AEGIS (80, 2) cycles
Memory bus	8-B wide
I/D TLB	4-way, 128 entries

and external memory. The size of the L2ID cache is controllable (256 KB, 1 MB, or 4 MB). The memory latency is controllable. In a typical processor, the memory latency is 18 cycles for the first access and 2 cycles for the successive accesses for DRAM burst transfers. In the proposed architecture, the latency is 40 cycles for the first access, according to the experimental evaluation in AEGIS[7]. In AEGIS, the latency is 80 cycles for the first access[7]. The simulation used the SPEC CPU2000[13] benchmarks. To capture the characteristics of the benchmarks in the middle of their computations, each benchmark component was simulated for 100 million instructions after skipping the first 1.5 billion instructions.

Figure 8 shows the Instructions per Cycle (IPC). Normal denotes the IPC of a typical processor. Proposed and AEGIS denote the IPC of the proposed architecture and AEGIS, respectively. Figure 9 also shows the L2ID cache mess rates.

The IPCs of the proposed architecture are always better than those of AEGIS. For high cache miss rates, the IPCs are improved by 60% in comparison to those of AEGIS. On the whole, the L2ID cache miss rates directly affect

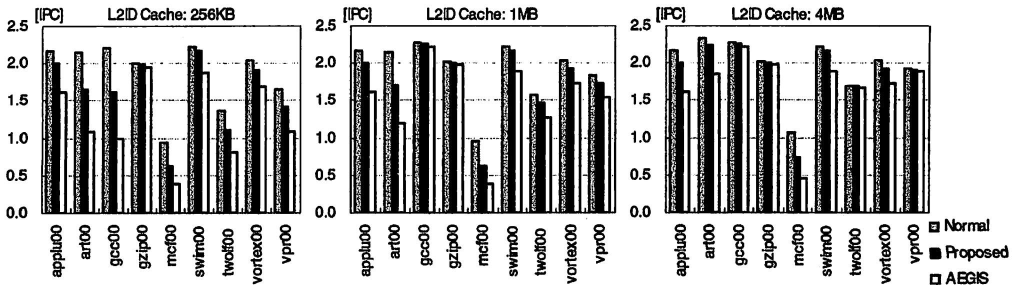

Fig. 8 The IPCs in SPEC CPU2000 benchmarks.

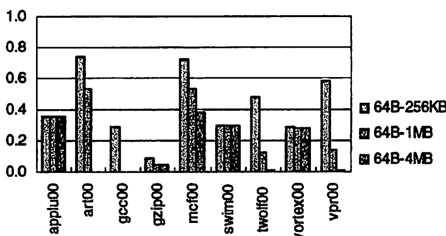

Fig. 9 The L2ID cache miss rates.

the IPCs. The reason is that when an L2ID cache miss occurs, the normal processor accesses memory, whose latency is high. In addition, the proposed architecture calculates Rijndael while AEGIS calculates a heavy IMHF. This appears to affect the execution performance directly.

5. Conclusions

This paper presented a processor-integrated approach to memory integrity as a fundamental technique for a secure computing infrastructure. The proposed architecture monitors the transactions between cache and external memory. The transactions are encrypted by using a high-speed block cipher, and then are summed up by XOR operations into a transaction history. This provides a high-speed architecture for the memory integrity verification.

A simple combination of a block cipher and XOR operations is not sufficient to prevent some clever attacks that utilize the characteristics of XOR operations. The previous work is resistant against these attacks by using a heavy IMHF. On the other hand, our proposed architecture has solved these problems by utilizing a special counting scheme for memory accesses while retaining the fast memory transaction capability.

In the simulations, the latency of Rijndael was pessimistically estimated as 40 cycles. However, recent typical Rijndael hardware implementations can calculate results within 12 cycles[14]. Using such circuits can hide the overhead related to verifying the memory integrity within the general memory access latency.

References

1) P. England, B. Lampson, J. Manferdelli, M. Peinado and B. Willman, "A Trusted Open Platform," *IEEE Computer Magazine*, Jul. 2003.
2) LaGrande Technology `http://www.intel.com/technology/security/`
3) Trusted Computing Group `https://www.trustedcomputinggroup.org/`
4) D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell and M. Horowitz, "Architectural Support for Copy and Tamper Resistant Software," *Proc. of the 9th Int.l Conference on Architectural Support for Programming Languages and Operating Systems*, pp.168-177, Nov. 2000.
5) D. Lie, C. Thekkath and Mark Horowitz, "Implementing an Untrusted Operating System on Trusted Hardware," *Proc. of the 19th ACM Symposium on Operating Systems Principles*, pp.178-192, Oct. 2003.
6) G. E. Suh, D. Clarke, B. Gassend, M. van Dijk and S. Devadas, "AEGIS: Architecture for Tamper-Evident and Tamper-Resistant Processing," *Proc. of the 17th Int.l Conference on Supercomputing*, Jun. 2003.
7) G. E. Suh, D. Clarke, B. Gassend, M. van Dijk and S. Devadas, "Efficient Memory Integrity Verification and Encryption for Secure Processing," *Proc. of the 36th Annual Int.l Symposium on Microarchitecture*, Dec. 2003.
8) G. E. Suh, C. W. O'Donnell, I. Sachdev, S. Devadas, "Design and Implementation of the AEGIS Single-Chip Secure Processor Using Physical Random Functions," *Proc. of the 32nd Int.l Symposium on Computer Architecture*, Jun. 2005.
9) D. Clarke, S. Devadas, M. van Dijk, B. Gassend and G. E. Suh, "Incremental Multiset Hash Functions and Their Application to Memory Integrity Checking," *Asiacrypt 2003*.
10) National Institute of Standards and Technology, "FIPS 180-2, Secure Hash Standard (SHS)", Aug. 2002.
11) J. Daemen and V. Rijmen, "AES Proposal: Rijndael," National Institute of Standards and Technology AES Proposal, Jun 1998.
12) D. Burger and T. M. Austin, "The SimpleScalar Tool Set, Version 2.0," *Technical Report, University of Wisconsin-Madison Computer Science Department*, 1997.
13) J. L. Henning. "SPEC CPU2000: Measuring CPU Performance in the New Millennium," *IEEE Computer*, Jul. 2000.
14) A. Satoh, "Unified Hardware Architecture for the Secure Hash Standard," *Embedded Cryptographic Hardware: Methodologies and Architectures, ISBN: 1594540128, NOVA Science*, Oct. 2004.